Transforming XACML policies into database search queries

Jasper Bogaerts *Bert Lagaisse* *Wouter Joosen*

Report CW 701, February 2017

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Transforming XACML policies into database search queries

Jasper Bogaerts *Bert Lagaisse* *Wouter Joosen*

Report CW 701, February 2017

Department of Computer Science, KU Leuven

Abstract

Application-level access control enforcement of complex policies is suffering from bad performance. This is especially true for search operations when the query results must be filtered by the application according to constraints of access control policies. One approach to reduce this overhead is to incorporate the access control policy in search queries on such data through query rewriting. This poses challenges to what can be expressed as part of such queries when complex policies must be taken into account, especially for expressive policy languages such as XACML. This paper proposes a transformation that converts attribute-based XACML policies to database queries while maintaining original policy semantics. This includes coping with XACML properties such as policy trees and many-valued logic. Our analysis verifies that the transformation leads to equivalent evaluation decisions and that it is a promising step towards policy-based database security.

Keywords : Access control, policy-based access control, databases, attribute-based access control, XACML, query rewriting.

Transforming XACML policies into database search queries

Jasper Bogaerts, Bert Lagaisse, and Wouter Joosen

iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
first.last@cs.kuleuven.be

Abstract. Application-level access control enforcement of complex policies is suffering from bad performance. This is especially true for search operations when the query results must be filtered by the application according to constraints of access control policies. One approach to reduce this overhead is to incorporate the access control policy in search queries on such data through query rewriting. This poses challenges to what can be expressed as part of such queries when complex policies must be taken into account, especially for expressive policy languages such as XACML. This paper proposes a transformation that converts attribute-based XACML policies to database queries while maintaining original policy semantics. This includes coping with XACML properties such as policy trees and many-valued logic. Our analysis verifies that the transformation leads to equivalent evaluation decisions and that it is a promising step towards policy-based database security.

Keywords: Access control, policy-based access control, databases, attribute-based access control, XACML, query rewriting

1 Introduction

Data security is an issue that is increasingly important, among others due to the growing popularity of big data systems. Growing amounts of data that are stored, retrieved and analyzed by such systems call for comprehensive security measures that can scale with the size of the data set on which they are enforced, especially for operations such as search.

In particular, access control is one such security measure for which the enforcement must scale with regard to the size of the data set. Access control restricts subjects (e.g., users) from performing actions on objects (i.e., resources) in a certain context. Access control constraints are specified in a policy, and enforcing expressive constraints can be problematic for search operations.

For example, consider an electronic document processing platform that generates, manages, and distributes business documents (e.g., invoices), and also is one of the three case studies that drove this research [2]. This application supports searching documents. However, when subjects do so, only documents to which they are entitled according to the access control policy should be returned.

One approach to cope with this is to describe access constraints as part of a database query by means of views [1, 17]. However, such an approach complicates the interface with the database [6] and involves the database administrators in the specification process, violating the separation of concerns principle [8].

To mitigate such issues, we propose the use of an externalized policy to determine the access constraints against the result of a search operation. One important technology that enables this is XACML [14]. XACML supports the expression of complex constraints that involve attributes associated with subjects, resources, actions and the environment (e.g., the current time). For example, it enables expressions such as *"subjects of the sales department can search documents shared with customers assigned to them"*. A naive approach to enable externalized policy enforcement for search with such constraints is to serially evaluate the access control policy against each element of the result set of the search operation, as illustrated in Figure 1. This approach does not scale with regard to size of the data set, because it involves evaluating complex expressions and fetching of attribute values for each item that results from the search, which typically leads to considerable overhead [7, 19].

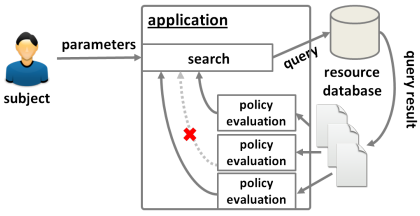

Fig. 1. A naive approach evaluates the policy for each item in the result set

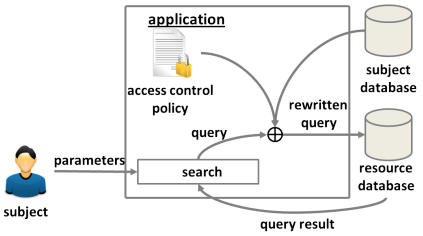

Fig. 2. Query rewriting combines search parameters with the transformed policy

Instead, we propose a query rewriting approach that combines the policy and search parameters into a compound query, as illustrated in Figure 2. First, we substitute all attributes corresponding to the subject, action and environment in the policy, because they remain fixed for the operation. Next, we transform the policy into a query and compose it with the search parameters.

Query rewriting can reduce scaling issues involved with serial evaluation, while maintaining the separation of concerns of a policy-based approach. However, in order to do this, a transformation of the policy must be performed. The result is a query expression that must be equivalent to the access control policy. In other words, every request that is denied for a certain subject, resource, action and environment for the policy evaluation must also not be returned as part of the result set of the query. This requires overcoming fundamental issues in how expressive policies such as XACML can be transformed to query constraints.

This paper introduces an approach capable of performing this transformation for a subset of XACML that is optimized for queries. Also, it verifies equivalence with policy evaluation and evaluates the complexity of the resulting expressions

with regard to the original policy. This provides a first step towards a query rewriting middleware that enables scalable access control for search operations.

The paper is organized as follows: Section 2 describes related work. Section 3 introduces the models of both the XACML language and query that we use as a basis for the transformation. Section 4 elaborates on the transformation approach. It also presents proof that this leads to equivalent access control enforcement. Section 5 analyzes the size complexity of the resulting query expression with regard to the original policy. Section 6 concludes the paper.

2 Related work

A lot of prior research has focused on database security [6]. In general, four approaches are related to the optimization of access control for search operations.

First, Fine-Grained Access Control (FGAC) techniques provide row-level security for databases by means of query rewriting techniques [1, 9, 17]. Typically, this involves extending the queries by means of policies that are specified in query language and may be described as part of views. However, because access constraints are described in a query language, they violate the principle of separation of concerns [8] because the database administrator must be involved in policy specification. Moreover, these technologies generally assume a two-tier architecture in which a subject directly queries the database. Contemporary applications are commonly designed in a multi-tier architecture. As a result, the applications perform queries on behalf of a subject, but are identified and secured as the users of the database [18]. This supports only coarse-grained policies that reason about application permissions[1]. Our approach, on the other hand, does enable fine-grained access control for multi-tier architectures.

Second, approaches exist that configure the access control system of the database based on an external policy. Notably, MyABDAC [11] compiles a XACML policy into an access control list. Mutti et al. [13] take a similar approach to extend SQLite for SELinux support. However, these approaches can have issues with large data sets and may require a modification of the DBMS.

Third, serial policy evaluation can be performed for each of the results of a database operation. Bouncer [15] is a system that takes this approach for the CPOL trust management system. However, this can lead to significant overhead when the size of the result set increases.

In this paper, we take a query rewriting approach that expresses the external access control policy as part of the search query. This retains separation of concerns, remains scalable with regard the data set size and does not require DBMS modification. Axiomatics Data Access Filter [5] supports a technology that applies query rewriting to an externalized XACML policy. However, they do not provide verification of equivalence of the original XACML policy and the query. Besides a transformation approach, this paper also verifies equivalence.

[1] Oracle's Virtual Private Database (VPD, [1]) supports queries that identify both application and subject, but still requires in-database policy specification.

Recently, Turkmen et al. [20] proposed a similar approach to ours to transform XACML policies in terms of boolean expressions for policy analysis. They defined a range of disjoint decision spaces in which access requests are divided (i.e., for permit, deny, indeterminate and not applicable requests). However, their approach cannot be used for this work because they define the *not applicable* decision space in terms of request membership of other decision spaces. In our approach, not applicable requests by default lead to a deny decision and non-inclusion of the relevant element in the result set. This ultimately leads to smaller query expressions because we only focus on recognizing permissive requests. Moreover, [20] does not verify equivalence with the original policy.

3 A model for XACML and database queries

This section describes a policy model that largely reflects the characteristics of the XACML language. It also introduces a model for database queries. These models are used throughout the remainder of the paper.

We do not consider special characteristics of XACML such as obligations and the extended indeterminate set, as we did not encounter a need for them in the context of search operations in our case studies. However, they constitute an interesting topic for future work.

3.1 XACML

In order to accomodate a transformation, we focus on XACML [14] as the policy language of the original policy. XACML provides an interesting language model that has been an inspiration to several other access control policy languages such as ALFA [4] and STAPL [12]. Also, XACML supports the specification in popular access control models such as RBAC [16] and ABAC [10]. The language model of XACML supports several features such as policy trees, many-valued logic and logical expressions that asses attribute comparisons.

Informally, XACML provides a tree-structured, attribute-based policy language in which policy components can themselves contain other policy components, thus forming a policy *tree* with rules as leaf elements[2]. We refer to them both as *policy elements*. Applicability of policy elements is determined during the evaluation, in which a *target* expression indicates applicability for a policy component, and a *condition* expression indicates applicability for a rule. We do not regard targets of rules, as they can be combined in a conjunction with the condition expression for the same evaluation result. If a rule is applicable, its corresponding decision (i.e., permit or deny) is taken into account. Expressions compare attributes assigned to subjects, resources, actions and environment with each other and concrete values in order to determine the applicability. Combining algorithms provide conflict resolution when multiple policy elements are applicable (e.g., first applicable, permit overrides).

[2] We do not regard policy *sets* as a separate entity type for this model and instead refer to them as policy components as well.

We define an expression as follows:

Definition 1. *An expression $e \in Expr$ is of the form*

$$e := a_1 \rhd val \mid a_1 \rhd a_2 \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid \neg e_1 \mid \varnothing$$

In which $a_1, a_2 \in Attr$ indicate attributes that need to be substituted with concrete values when the expression is evaluated, $val \in V$ a concrete value part of the set of all possible concrete values and $e_1, e_2 \in Expr$ other expressions. Operator $\rhd$ indicates how values can be compared (e.g., $<, \leq, =$).

Policy elements are specified more formally as follows:

Definition 2. *A policy element* p *can be of two forms:*

- *A tuple $(t, p_c, comb)$ indicates a **policy component** with target expression $t \in Expr$ part of the set of all valid expressions, a totally ordered, non-empty set of child elements p_c and combining algorithm comb.*
- *A tuple $(cond, eff)$ indicates a **rule element** with a condition expression $cond \in Expr$ and an effect $eff \in \{permit, deny\}$.*

We define `permit(p)` and `deny(p)` as the sets of all rules that have a permit and deny effect, respectively, for a policy component p. Also, `children(p)` reflects the totally ordered set of all rules of the policy component p. Boolean function `rule(p)` indicates if a policy element p is a rule. Functions `cond(r)` and `target(p)` extract the condition and target expressions of a rule and policy component, respectively. Similarly, `eff(r)` indicates the effect of a rule and `comb(p)` the combining algorithm of a policy. Finally, we denote P the set of all valid policy elements. Obligations and the extended indeterminate set of XACML 3.0 (containing decisions that indicate what should have been returned if no error had occurred, such as $Indeterminate\{P\}$) are not considered in this model because we did not encounter a need for them in our case studies.

To illustrate this, Figure 3 considers a schematic example of a policy for an electronic document processing platform. Rules r_1, r_2, r_3 and r_4 have conditions of the form "*resource.creator* == *subject.id*" an ownership rule, or "*resource.type* == '*catalog*'". Subject attributes are substituted prior to the transformation because a single subject performs the search. Policy components p_1 and p_2 have an empty target that is always applicable, denoted by $\varnothing$.

In order to evaluate a policy, the applicable child elements must be evaluated and their evaluation outcomes combined using a combining algorithm. Applicability is determined by evaluating target and condition expressions. For this, we define function `evalExpr`.

Definition 3. *An expression evaluation `evalExpr`: $Expr \times \alpha \rightarrow \{true, false\}$ is a function that takes as input a mapping $\alpha : Attr \rightarrow V$ that projects all attributes $a_i \in Attr$ of the expression onto concrete values $val \in V$ and evaluates whether the expression is true when the concrete values are taken into account.*

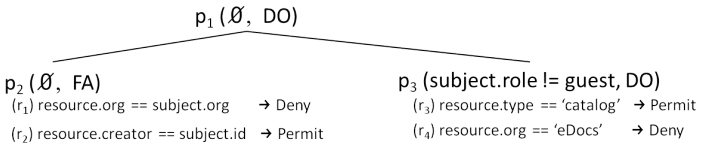

Fig. 3. The policy includes policy components p_1, p_2 and p_3 with a first applicable (FA) and deny overrides (DO) combining algorithms and rules (r_1, r_2 r_3 and r_4). Unlike p_1 and p_2, p_3 is not applicable for guests. We did not include any actions for brevity.

For example, rule r_3 in Figure 3 is applicable when the evaluated resource in the evaluation context α is a *catalog*. Otherwise, `evalExpr` evaluates to **false**.

For expressions, an *evaluation context* $\alpha : Attr \rightarrow V$ enables substitution of subject, action and environment attributes to values. We assume that every context α is always complete, i.e., that every attribute is mapped to a value.

Alternatively, `evalExpr` can also lead to an *Indeterminate* decision, e.g., whenever an attribute could not be retrieved. We propagate this decision to the result of the policy element evaluation by `evalPolicy`.

Definition 4. *Function* `evalPolicy`: $P \times \alpha \rightarrow \{Permit, Deny, NotApplicable, Indeterminate\}$ *takes as input a context* $\alpha : Attr \rightarrow V$ *and evaluates all policy elements to come to a decision* $d \in \{Permit, Deny, NotApplicable, Indeterminate\}$. *The decision process for a function* $evalPolicy(p, \alpha)$ *with evaluation context* α *and* $p \in P$ *is determined based on four aspects:*

1. *Indeterminate decisions are returned when problems occur that impede evaluation (e.g., network problems, invalid policy or attribute retrieval issues).*
2. *If p is a rule, then* $evalPolicy(p, \alpha) = eff(p)$ *if its condition is satisfied. If the condition is not satisfied, then it evaluates to not applicable (NA).*
3. *If p is a policy component, and each child element evaluates to NA, then p evaluates to NA. It also evaluates to NA if its target evaluates to false.*
4. *Otherwise, if p is a policy, the result of* $evalPolicy(p, \alpha)$ *is determined by the combining algorithm* $comb(p)$.

We consider the transformation for three common combining algorithms: first applicable (`FA`), permit overrides (`PO`) and deny overrides (`DO`). Supplementary to the aspects above, they influence the decision as follows:

First Applicable. This algorithm considers the first policy element that is applicable and returns the result of its evaluation. Formally, for $p_1, p_2, ..., p_n \in children(p)$ the child elements of a policy p in which $p_1 < p_2 < ... < p_n$, the decision of a policy component p under context α with $comb(p) = FA$ is defined as $evalPolicy(p, \alpha) = e \Leftrightarrow [\exists p_i \in children(p) : evalPolicy(p_i, \alpha) = e \ \wedge \ (\forall p_j \in children(p) : evalPolicy(p_j, \alpha) \neq NA \Rightarrow p_j > p_i)]$, with $e \in \{Permit, Deny\}$.

Deny Overrides. In this algorithm, any child element that evaluates to a *deny* decision overrides any other decisions. More formally, $evalPolicy(p, \alpha) =$

$Deny \Leftrightarrow \exists c \in children(p) : evalPolicy(c, \alpha) = Deny$. Such a policy evaluates to a permit decision if a child element evaluates to this and no deny decision is encountered, described as $evalPolicy(p, \alpha) = Permit \Leftrightarrow \exists c \in children(p) : evalPolicy(c, \alpha) = Permit \wedge \neg \exists c \in children(p) : evalPolicy(c, \alpha) = Deny$.

Permit Overrides. In this algorithm child elements that evaluate to *permit* override other decisions: $evalPolicy(p, \alpha) = Permit \Leftrightarrow \exists c \in children(p) : evalPolicy(c, \alpha) = Permit$ and $evalPolicy(p, \alpha) = Deny \Leftrightarrow \exists c \in children(p) : evalPolicy(c, \alpha) = Deny \ \wedge \ \neg \exists \ c \in children(p) : evalPolicy(c, \alpha) = Permit$.

3.2 Database queries

Contrary to XACML policies, database queries consist of expressions that compare table cells with concrete values or each other to determine whether an element in a row should be selected. This paper considers queries of the form `SELECT * FROM table WHERE expr` in which `expr` specifies constraints to filter out the table contents by. The query returns all rows that satisfy `expr`.

Database expressions are similar to policy expressions. Consider T the set of all tables in a database schema. Function `columns(t)` denotes the set of columns that exists for any table $t \in T$. In this paper, we disregard schemaless databases.

Definition 5. *A database expression is a tuple (t, e) in which $t \in T$ describes the table of the elements, and e is a query expression of the form $e := t.c_1 \triangleright val \ \mid \ t.c_1 \triangleright t.c_2 \ \mid \ e_1 \wedge e_2 \ \mid \ e_1 \vee e_2 \ \mid \ \neg e_1$, with e_1, e_2 other query expressions, $t.c_i$ indicating the column c_i of a table t, val a concrete value and comparison operator $\triangleright$ (e.g., $<, \leq, =$).*

We treat each table row as separate resource described by the contents in its corresponding columns. Columns are considered the *attributes* of the resource.

Note that some resource attributes could be part of another table $t_2 \in T$, and would require joins to address them. For brevity, we do not consider this here but this does not limit expressiveness of the policy in terms of the query.

4 Transformation

This section elaborates on the transformation approach. The approach gradually transforms the policy to a *flat* policy, i.e., a single policy component with only rules as children. Next, it converts the conditions of its permit rules into a query expression that is composed with search parameters. First, we introduce the concept of a *well-defined policy*, which is a necessary prerequisite for the transformation. Next, we detail the approach further. We also verify correctness for the policy flattening.

The transformation approach requires a well-defined policy because there always needs to be a default decision over whether a request is permitted or not.

Definition 6. *A well-defined policy p is a policy for which each evaluation results in a permit or deny decision. More formally: p is well-defined* $\Leftrightarrow \forall \alpha : evalPolicy(p, \alpha) \in \{Permit, Deny\}$.

To support this, we assume that an evaluation error that would otherwise result in an `indeterminate` decision is handled by the underlying application platform. This includes, among others, issues with missing attributes which due to the transformation might lead to a query execution error. This does not support the use of extended indeterminate decisions that are specified in XACML 3.0 [14] which can lead to an evaluation decision (i.e., permit or deny) depending on the source of the indeterminate decision, which constitutes an interesting topic for future work. In the case studies that drove this research, we did not encounter a need for this and instead, any errors should lead to search query being aborted.

If this is taken into account, any policy can be extended to a well-defined policy given a default decision to override *NA* decisions. For this work, we deny such requests. This is done by constructing a parent policy p_{FA} with an empty target, an FA algorithm, and two children policy elements. The first child is the original policy component, and the second element always evaluates to `Deny`.
To transform a well-defined policy p to a flat policy, we iterate over two steps.

1. Every policy component with only rules as children is converted to a policy component with PO algorithm. The combination of its rules must reflect the same evaluation result as the original policy.
2. This step regards policy components that combine other PO policy components with rules as children, i.e., the grandparent policies to the rules of PO policy components. It converts each of them to a policy component with PO algorithm and only rules as children. This enables flattening the policy.

We iterate over these steps until we have a flat policy component with PO algorithm that contains only rules as children. The transformation maintains equivalence over `evalPolicy`, meaning that for every evaluation context α, the evaluation result for both original and flat policy should be the same. Hence, the permit rules of the flat policy will reflect the only cases in which the evaluation can evaluate to permit, and thus for which any item of the result set of a search should be returned. Consequently, the conditions of all permit rules are combined into the query expression with the original search parameters. Note that this differs from the goal of [3], which determines the attribute sets for decisive evaluation, and from [20] which constructs decision spaces for policy analysis.

An example of the two flattening steps is illustrated in Figure 4. The figure shows an example transformation of a policy tree into a *flat* policy that contains only rules as children. The permit rules of the flat policy are then combined into a query expression.

The remainder of this section elaborates on the flattening steps and demonstrates equivalence with regard to evaluation on the original policy. Next, it elaborates on the transformation of the flat policy to a database query.

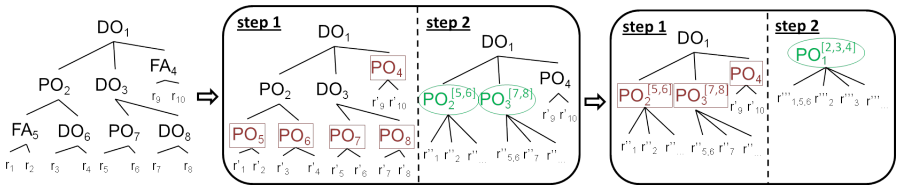

Fig. 4. Example of a policy tree that is gradually transformed into a flat policy as a first part of the transformation. The results of the first step are illustrated in red (squared), results of the second step in green (circled).

4.1 Step 1: Converting to PO policies

In the first step, each policy component of a policy that has only rules as children is converted to a policy component with PO algorithm. This is also shown in Figure 5.

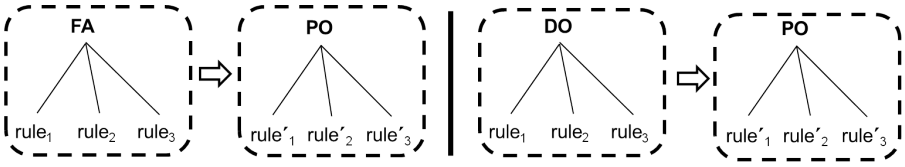

Fig. 5. The first transformation step converts every policy component with only rules as children to a policy component with PO algorithm.

We distinguish between policy components with a DO and FA algorithm to describe the conversion. We specify a transformation function $\Theta : P \rightarrow P$ that converts a policy component to one with PO algorithm. Since policy components with PO algorithm already satisfy this goal, we do not consider them here.

Deny overrides. To convert a policy component p with $comb(p) = DO$ to a PO policy component, consider that every deny rule $r_{d_i} \in deny(p)$ is *decisive*, i.e., it overrides other decisions if both r_{d_i} and p are applicable under a evaluation context α. We specify the transformation function Θ_{DO} as follows:

$$\Theta_{DO}(p) := p' \text{ with } \begin{cases} children(p') = \{\Phi_{DO}(r, p) \mid r \in children(p)\} \\ target(p') = target(p) \\ comb(p') = PO \end{cases}$$

In which Φ_{DO} is a helper function to transform the rules that ensures permit rules can only be applicable if no deny rule is applicable. It does this by concatenating negations of conditions of the deny rules for each permit rule, more

formally:

$$\Phi_{DO}(r_i, p) := r_i' \text{ with } \begin{cases} cond(r_i') = \begin{cases} cond(r_i) \wedge (\bigwedge_{r_j \in deny(p)} \neg cond(r_j)), & \text{if } r_i \in permit(p) \\ cond(r_i), & \text{if } r_i \in deny(p) \end{cases} \\ eff(r_i') = eff(r_i) \end{cases}$$

For example, policy p_3 of Figure 3 would generate a novel permit rule r_3' with condition *"resource.org != 'eDocs' and resource.type = 'catalog' "*. The original deny rule r_4 would also be included in transformed policy $\Theta_{DO}(p_3)$. Note that p_1 is not transformed in this step as it has policy components as children.

Under this transformation, it suffices to demonstrate that *for each* evaluation context α, $evalPolicy(p, \alpha) = d \Rightarrow evalPolicy(\Theta_{DO}(p), \alpha) = d$ with $d \in \{Permit, Deny, NA\}$. We refer to this property as *equivalence*. In this step, this property holds when $\neg rule(p)$ and $\forall c \in children(p) : rule(c)$.

Proof. In order to prove equivalence, we distinguish between three cases. First, we show that for all α, $evalPolicy(p, \alpha) = NA \Rightarrow evalPolicy(\Theta_{DO}(p), \alpha) = NA$. Second, $evalPolicy(p, \alpha) = Permit \Rightarrow evalPolicy(\Theta_{DO}(p), \alpha) = Permit$. Third, $evalPolicy(p, \alpha) = Deny \Rightarrow evalPolicy(\Theta_{DO}(p), \alpha) = Deny$. Since we argue this for all α, equivalence is proven. In the first case, if $evalPolicy(p, \alpha) = NA$ then either $evalExpr(target(p), \alpha) = false$, or else $\neg\exists r \in children(p) : evalExpr(cond(r), \alpha) = true$. The original target is retained under the transformation, so an inapplicable target remains so. When there is no applicable rule in the original policy, then this will also hold under the transformation because for permit rules the condition depends on the original conditions of permit rules which can never be true. For deny rules, the original condition is retained. As a result, $evalPolicy(\Theta_{DO}(p), \alpha) = NA$. The second case holds because none of the deny rules $r_d \in deny(p)$ are applicable. If they were, then the evaluation would lead to a deny decision. Since $evalExpr(cond(r_d), \alpha) = false$ for all r_d, the second clause of permit rules in Φ_{DO} are true under α since it is a conjunction of negated deny rule conditions. Because $evalPolicy(p, \alpha) = Permit$, $\exists r_p \in permit(p) : evalExpr(cond(r_p), \alpha) = true$. As a result, at least $\Phi_{DO}(r_p)$ must be applicable, and $evalPolicy(\Theta_{DO}(p), \alpha) = Permit$. The third case holds because there must be at least one rule $r_d \in deny(p)$ which is applicable. As a consequence, none of the permit rules from the transformation can be applicable because of the second clause of Φ_{DO}. Moreover, the original deny rules are included in Φ_{DO} and therefore $evalPolicy(\Theta_{DO}(p), \alpha) = Deny$. □

Consequently, we can transform any policy component with only rules as children and a DO algorithm to an equivalent policy with PO algorithm.

First applicable. Given a policy component p with $comb(p) = FA$ that must be converted to a PO policy component. Let $r_1, r_2, ..., r_n \in children(p)$ be an ordered list of rules with $r_1 < r_2 < ... < r_n$. For a FA evaluation, a rule $r_i \in children(p)$ is *decisive* under α if $evalExpr(cond(r_i), \alpha) = true$ and $\neg\exists r_j \in children(p) : r_j < r_i \wedge evalExpr(cond(r_j), \alpha) = true$. The transformation incorporates the decisiveness of deny rules into permit rules by means of a

transformation function Φ_{FA}. We specify this function as

$$\Phi_{FA}(r_i, p) := r_i' \text{ with } \begin{cases} cond(r_i') = \begin{cases} cond(r_i) \wedge (\bigwedge_{r_j \in deny(p): r_j < r_i} \neg cond(r_j)), & \text{if } r_i \in permit(p) \\ cond(r_i), & \text{if } r_i \in deny(p) \end{cases} \\ eff(r_i') = eff(r_i) \end{cases}$$

Function Φ_{FA} maintains the original deny rules while it includes the negation of conditions of prior deny rules in permit rules. We define Θ_{FA} as follows:

$$\Theta_{FA}(p) := \text{p' with } \begin{cases} children(p') = \{\Phi_{FA}(r, p) \mid r \in children(p)\} \\ target(p') = target(p) \\ comb(p') = PO \end{cases}$$

For example, the transformation of policy p_2 from Figure 3 includes deny rule r_1, and permit rule $\Phi_{FA}(r_2) = r_2'$ has condition *"resource.org != 'LargeBank' and resource.creator == 43"* if the subject that performed the request is not a guest, has identifier *43* and is affiliated with organization *LargeBank*.

Under this transformation, equivalence between p and $\Theta_{FA}(p)$ holds.

Proof. We again consider the three cases as in the previous proof. For the first case, we argue that Φ_{FA} again includes all of the original conditions, so not applicable rules remain so. Also, the original target is included. As a result, equivalence holds for the NA decision. For the second case, if $evalPolicy(p, \alpha) = Permit$, $\exists r_i \in permit(p) : evalExpr(cond(r_i), \alpha) = true \wedge (\neg \exists r_j \in children(p) : evalExpr(cond(r_j), \alpha) = true \wedge r_j < r_i)$. Because $comb(\Theta_{FA}(p)) = PO$ it suffices to prove equivalence under permit rules of the transformation. From the definition of Φ_{FA} follows that permit rules include conjunctions of negations of deny rule conditions. For every $r_j < r_i$, $evalExpr(cond(r_j), \alpha) = false$ because they are not applicable. As a result, the condition of at least one permit rule $r_p \in permit(p)$ remains applicable, because every deny rule before it does not have an applicable condition, and the condition of every $r_j < r_p$ is negated and conjuncted to the condition of r_p. If $evalExpr(cond(\Phi_{FA}(r_p)), \alpha)$ would not be true, this would contradict the premise that r_p is the first applicable rule of p. The third case is similar to the second one. Since $evalPolicy(p, \alpha) = Deny$, there exists a rule $r_d \in deny(p)$ for which $evalExpr(cond(r_d), \alpha) = true$ and for which $\forall\, r \in children(p) : r < r_d \Rightarrow evalExpr(cond(r), \alpha) = false$. Consequently, for each permit rule $r_p' \in permit(\Theta_{FA}(p))$, there is a permit rule $r_p \in permit(p)$ for which either (a) $r_p < r_d$, or (b) $r_p > r_d$. In case of (a), $evalExpr(cond(r_p), \alpha) = false$ because this would otherwise violate the requirement for r_d to be decisive. In case of (b), the transformation r_p' cannot be applicable as it includes a negation of the condition of r_d as part of its clause, which cannot be satisfied. As a result, $evalPolicy(\Theta_{FA}(p), \alpha) = Deny$. □

4.2 Step 2: Combining policy components

The second step of the transformation flattens each policy component that has as children other policy components with only rules and a PO algorithm. More formally, this requires for a policy component p that $\forall p' \in children(p) : rule(p') =$

$false \wedge comb(p') = PO \wedge (\forall r \in children(p') : rule(r) = true)$. The transformation incorporates the rules of the children into a novel, flattened policy component that has a PO combining algorithm. This is also shown in Figure 6.

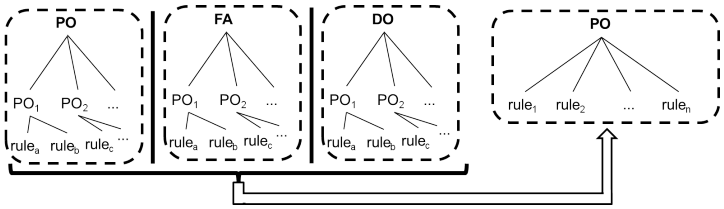

Fig. 6. The second transformation step combines rules of child policies into a novel, flattened PO policy component. We support this for PO, DO and FA algorithms.

A rule is applicable only if both its condition as well as the target of its parent policy component are true. Therefore, we first compose the target of each child policy component with the condition of its rules. Next, we perform a transformation for which we again distinguish between the combining algorithm of the parent policy component. We will only consider DO and FA transformations here, as the PO transformation includes the original rules of the assimilated policy component.

Deny overrides. To transform a policy component p with $comb(p) = DO$ and child policies $p_{c_1}, p_{c_2}, ..., p_{c_n} \in children(p)$ to a policy component with only rules as children and PO algorithm, we define function Δ_{DO}. This function uses helper function Ψ_{DO} which generates a single permit rule. This rule must never be applicable if one of the child policies leads to a deny decision. To ensure this, for every child policy component $p_{c_i} \in children(p)$ we must thus make sure that $\varphi(p_{c_i}) := [(\bigvee_{r_p \in permit(p_{c_i})} cond(r_p)) \vee (\bigwedge_{r_d \in deny(p_{c_i})} \neg cond(r_d))]$ holds. In other words, either a permit rule is applicable or no deny rule is applicable.

If we assume `permits(p)` the set of all permit rules of all child policy components of p, we can specify helper function Ψ_{DO} as

$$\Psi_{DO}(p) := r' \text{ with } \begin{cases} cond(r') = (\bigvee_{r_p \in permits(p)} cond(r_p) \wedge \bigwedge_{p_c \in children(p)} \varphi(p_c) \\ eff(r') = permit \end{cases}$$

The rule of Ψ_{DO} is only applicable if at least one child element evaluates to permit (first clause), and no other child elements evaluate to deny (second clause). Assuming `denies(p)` the set of all deny rules of all child policy components of p, we can now define transformation function Δ_{DO} as:

$$\Delta_{DO}(p) := p' \text{ with } \begin{cases} children(p') = \{\Psi_{DO}(p)\} \cup \{r \mid r \in denies(p)\} \\ target(p') = target(p) \\ comb(p') = PO \end{cases}$$

This transformation function leaves the evaluation of the transformed policy equivalent to the original policy.

For example, the transformation of policy p_1 from Figure 3 includes the original deny rules, and constructs a rule $r'_{2,3}$ with condition *"[(resource.creator == 43 and resource.org != 'LargeBank') or (resource.type == 'catalog' and resource.org != 'eDocs')] and [(resource.creator == 43 and resource.org != 'LargeBank') or resource.org != 'LargeBank'] and [(resource.type == 'catalog' and resource.org != 'eDocs') or resource.org != 'eDocs']"*. This can be normalized to *"(resource.creator == 43 or resource.type == 'catalog') and resource.org != 'LargeBank' and resource.org != 'eDocs' "*.

Proof. We prove equivalence by demonstrating the evaluation of the permit rule constructed by $\Psi_{DO}(p)$. Function Ψ_{DO} offers several observations:

1. If any of the child policies evaluates to permit under α, then the first clause of $\Psi_{DO}(p)$ will evaluate to true, since this is a disjunction of conditions of the original permit rules of which at least one must be applicable.
2. If all child policies evaluate to permit under α then the second clause of Ψ_{DO} evaluates to true, since this would mean that at least one rule $r_p \in permit(p_c)$ is applicable for each child policy p_c and the second clause includes a disjunction of these conditions for each child policy.
3. If all child policies are not applicable for α, then the second clause of Ψ_{DO} evaluates to true, since $\forall r_d \in children(p_c) : evalExpr(cond(r_d), \alpha) = false$ for all $p_c \in children(p)$ and by negation this satisfies the second clause.
4. If there exists a child policy component $p_D \in children(p)$ that evaluates to deny, then the second clause of Ψ_{DO} is false, since $\exists r_d \in deny(p_D) : evalExpr(\neg cond(r_d), \alpha) = false$ for that policy p_D and none of the conditions of the permit rules evaluate to true.

To prove equivalence, we again consider the three possible outcomes.

First, if $evalPolicy(p, \alpha) = NA$, then observation (3) argues that the second clause of Ψ_{DO} is true. However, the first clause of Ψ_{DO}, which is in conjunction with the satisfied clause, is false because no permit rule is applicable under α.

Second, if $evalPolicy(p, \alpha) = Permit$, there are two possible cases. Either (a) all of the child policies evaluate to permit, or (b) at least one of the child policies evaluates to permit and the other policies are not applicable. In case of (a), we conclude from observations (1) and (2) that equivalence holds under any α. In case of (b), it suffices to show from observations (2) and (3) that the second clause is true, as observation (1) ensures that the first clause is always true. Since for each child policy $\forall r_d \in children(p_c) : evalExpr(cond(r_d), \alpha) = false$ if it evaluates to not applicable, and $\exists r_p \in permit(p_c) : evalExpr(cond(r_p), \alpha) = true$ if it evaluates to permit. Hence, if a policy component p_c evaluates to permit or not applicable, $\Psi_{DO}(p)$ evaluates to permit for the evaluation context α.

Third, if $evalPolicy(p, \alpha) = Deny$, there exists at least one child policy $p_D \in children(p)$ for which $evalPolicy(p_D, \alpha) = Deny$. Hence, observation (4) holds and $\Psi_{DO}(p)$ is not applicable. Since the original deny rules are included in Δ_{DO}, at least one deny rule is applicable and Δ_{DO} evaluates to deny.

As a result, equivalence holds for every evaluation under any context α. □

First applicable. For the transformation of a policy component p with child policies $p_{c_1}, p_{c_2}, ..., p_{c_n} \in children(p)$, $comb(p_{c_i}) = PO$ and $comb(p) = FA$ to a policy component with only rules as children and PO algorithm, we define function Δ_{FA}. We note $p_{c_i} < p_{c_{i+1}}$ for $i > 0$ and $i < n$ if a policy component p_{c_i} precedes $p_{c_{i+1}}$ for the evaluation. We first specify helper function Ψ_{FA} to define Δ_{FA}. For this, we denote `pre(r)` $:= \{r_i \in deny(p_{c_i}) \mid p_{c_i} < p_c \; with \; r \in children(p_c)\}$.

$$\Psi_{FA}(r) := r' with \begin{cases} cond(r') = \begin{cases} cond(r) \wedge \bigwedge_{r_d \in pre(r)} \neg cond(r_d), & \text{if } \mathit{eff}(r) = permit \\ cond(r), & \text{if } \mathit{eff}(r) = deny \end{cases} \\ \mathit{eff}(r') = \mathit{eff}(r) \end{cases}$$

This helper function takes into account any deny rules that might be decisive prior to permit rules of latter policies. It enables us to define function Δ_{FA} as

$$\Delta_{FA}(p) := p' with \begin{cases} children(p') = \{\Psi_{FA}(r, p_c, p) \mid r \in children(p_c), p_c \in children(p)\} \\ target(p') = target(p) \\ comb(p') = PO \end{cases}$$

The evaluation of $\Delta_{FA}(p)$ is equivalent to that of original policy p.

Proof. We again consider the three outcomes for $evalPolicy(p, \alpha)$.

If $evalPolicy(p, \alpha) = NA$, then $\forall r \in children(p_c) : evalExpr(cond(r), \alpha) = false$ with $p_c \in children(p)$. Consequently, rules constructed by Ψ_{FA} are not applicable and $evalPolicy(\Delta_{FA}(p), \alpha) = NA$.

If $evalPolicy(p, \alpha) = Permit$, then $\exists p_P \in children(p) : evalPolicy(p_P, \alpha) = Permit \wedge (\forall p_c \in children(p) : p_c < p_P \Rightarrow evalPolicy(p_c, \alpha) = NA)$. Thus, $\exists r_p \in permit(p_P) : evalExpr(cond(r_p), \alpha) = true$ the first applicable rule of p_P and $\forall p_c < p_P : \neg\exists r_d \in deny(p_c) : evalExpr(cond(r_d), \alpha) = true$. Hence, Ψ_{FA} builds at least one applicable permit rule, so $evalPolicy(\Delta_{FA}(p), \alpha) = Permit$.

If $evalPolicy(p, \alpha) = Deny$, then $\exists p_D \in children(p) : evalPolicy(p_D, \alpha) = Deny$ and also $\exists r_d \in deny(p_D) : evalExpr(cond(r_d), \alpha) = true$ a first applicable rule of p_D and $\neg\exists p_c \in children(p) : p_c < p_D \wedge evalPolicy(p_c, \alpha) \neq NA$. It suffices to demonstrate that Ψ_{FA} (a) generates at least one applicable deny rule, and (b) does not generate any applicable permit rule. Proposition (a) follows from the deny rules, which are included verbatim with Ψ_{FA} for all child policies. For (b), suppose $r'_p \in permit(\Delta_{FA}(p))$ a permit rule generated with Ψ_{FA}. Since an applicable $r_d \in deny(p_D)$ also exists, for r'_p to be applicable, it must be true that it was generated from a condition of $r_p \in permit(p_P)$ with $p_P \in children(p)$ for which $p_P < p_D$. Otherwise, the second clause of Ψ_{FA} for permit rules would never be true. However, this would violate the premise that $evalPolicy(p, \alpha) = Deny$. As a result, $evalPolicy(\Delta_{FA}(p), \alpha) = Deny$. □

4.3 Transforming a flat policy to a database query

We iterate over the previous two steps of the transformation until we obtain a single policy component p for which $\forall \; el \in children(p) : rule(el) = true$. This is a *flat* policy. The last part of the transformation converts the child rules of the flat policy p to a database query clause. In order to do this, each of the

permit rules $r_p \in permit(p)$ is transformed by function ω. This is done for a table t which is the same table on which the search query is performed, and hence reflects the resources to be protected.

$$\omega(p,t) := q \; with \begin{cases} expression(q) = \bigvee_{r \in permit(p)} \tau(cond(r)) \\ table(q) = t \end{cases}$$

With $t \in T$ the database schema and a function τ that maps the referred attributes to the database columns of t. Such a mapping should be specified by the security administrator together with the policy.

The resulting query will only select the rows that satisfy the given policy p. The equivalence of this policy was demonstrated in the previous section. Because we convert a PO policy to a query expression, satisfying any of the clauses will suffice to include a row. Since p is decisive, it always evaluates to either a permit or a deny decision. This translates to a `true` evaluation of the elements of the query that would also lead to a permit decision for p, and `false` for elements that do not. Consequently, rows that do not satisfy the condition would be rejected by the policy evaluation and are also not returned by the query.

As indicated in Section 4.2, the transformation of the policy from Figure 3 results in a flat policy with three rules. This includes the two original deny rules r_1 and r_4, and permit rule $r'_{2,3}$ with condition *"(resource.creator == 43 or resource.type == 'catalog') and resource.org != 'LargeBank' and resource.org != 'eDocs' "* for a subject which is not a guest, is affiliated with *LargeBank* and has identifier *43*. The condition of rule $r'_{2,3}$ is directly translated to a query, mapping the resource attributes to the corresponding table columns in the process.

5 Complexity analysis

Figure 4 illustrated that the transformation approach iterates two times for a policy depth of three. In general, the number of iterations over the two flattening steps of the transformation is $d-1$, with d the *depth* of a policy tree (i.e., the maximum number of ancestor components of any rule in the policy).

Transformation of a policy tree to an equivalent flat policy may introduce redundancy with regard to the original conditions extracted from its rules. To analyze this, we again consider the impact of the two flattening steps to determine how many times an expression is repeated to retain semantic equivalence. We denote $|R_P(p)|$ and $|R_D(p)|$ the size of the set of permit and deny rule conditions of a policy component p, respectively. For the first step, we conclude:

- Transformation from a DO algorithm is possible to a single permit rule with condition $[\bigwedge_{r_d \in deny(p)} \neg cond(r_d)] \wedge [\bigvee_{r_p \in permit(p)} cond(r_p)]$. This is equivalent to the combination of permit rules from Θ_{DO}. At most $|R_P(p)| + |R_D(p)|$ expressions are combined to reflect the permit rule, and hence there is no redundancy.
- Transformation from a FA algorithm is similar. Conditions of the original deny rules can be interspersed in the conjunction to become a single permit rule with a condition of at most $|R_P(p)| + |R_D(p)|$ compounded expressions.
- A PO algorithm element is not transformed as it satisfies the goal of this step. The permit rules thus maintain the original $|R_P(p)|$ expressions.

As a result, the first flattening step does not by itself introduce any redundancy.

Analyzing redundancy for the second step is not as straightforward, as it requires reasoning in terms of the depth of the policy tree. To identify the worst-case scenario, we start by analyzing the case in which each ancestor along the path of the policy tree contains the same combining algorithm.

- Transformation from a DO algorithm requires that the conditions of permit rules of child policies are included both in disjunction and in a conjunction with the negation of conditions of their deny rules. Consequently, each condition of a permit rule is included twice for each policy component with DO algorithm. This results in a worst-case redundancy of $2^d \times |R_P| + (2^d - 1) \times |R_D|$.
- Transformation from a FA algorithm can again be based on the interspersion of the deny rule conditions between the permit rule conditions. Doing so, this transformation leads to at most $|R_P(p)| + |R_D(p)|$ compounded expressions.
- Transformation from a PO algorithm includes only the original rules. Hence, there is no redundancy here and at most $|R_P(p)|$ expressions are included.

This analysis shows that FA and PO algorithms do not introduce redundancy. However, DO algorithms may lead to significant redundancy. Practical experience from our case studies (among others, [2]) indicates that the policy depth typically does not grow large (at most 5 in our case studies), and that this includes other combining algorithms along the path. More importantly, in a scenario with subsequent DO ancestors in a policy tree, duplicate conditions cancel each other out, leading to no redundancy in that case. For instance, consider again the example for the DO algorithm transformation in Section 4.2. Here, because the child element was transformed from a DO algorithm as well, the expressions cancel each other out and result in no redundancy. In general, this is true for both FA and DO algorithms that are transformed under a parent DO. This leads to a worst-case redundancy of $2^{\lfloor d/2 \rfloor} \times |R_P| + (2^{\lfloor d/2 \rfloor} - 1) \times |R_D|$ when DO and PO algorithm components are interspersed along an ancestor path. Consequently, normalization may reduce the worst case effects significantly and further analysis for this is an interesting topic for future work.

6 Conclusion

In this paper, we have analyzed a transformation approach that supports conversion of a tree-structured, attribute-based policy language with four-valued logic (such as XACML) to an expression that can be combined together with a database search query to enforce access control. We have proven that this transformation leads to an equivalent evaluation, i.e., that it yields the same evaluation result as a regular policy evaluation. In future work, we will implement this and perform a thorough analysis on the performance of this approach.

The transformation only leads to significant redundancy in expressions when policy components with a deny overrides combining algorithm are involved. However, inefficient queries can be further reduced by case-specific measures and our case studies indicate that they would only involve a limited practical overhead. Nevertheless, we believe that developing measures to optimize this further constitutes an interesting topic for future work. We are convinced that this work is an important step towards support for policy-based access control for databases.

References

1. Oracle Virtual Private Database (VPD). http://docs.oracle.com/cd/B28359_01/network.111/b28531/vpd.htm. Accessed: 2015-09-02.
2. Reference withheld to preserve the anonymity of the authors. Technical report.
3. A. Anderson. Evaluating XACML as a policy language. Technical report, Technical report, OASIS, 2003.
4. Axiomatics. ALFA. http://www.axiomatics.com/solutions/products/authorization-for-applications/developer-tools-and-apis/192-axiomatics-language-for-authorization-alfa.html. Accessed 2016-02.
5. Axiomatics. Axiomatics Data Access Filter (ADAF). http://www.axiomatics.com/solutions/products/authorization-for-databases/197-axiomatics-data-access-filter-adaf.html. Accessed: 2016-02-25.
6. E. Bertino and R. Sandhu. Database security-concepts, approaches, and challenges. *Dependable and Secure Computing, IEEE Transactions on*, 2(1):2–19, 2005.
7. A. D. Brucker and H. Petritsch. Idea: efficient evaluation of access control constraints. In *Engineering Secure Software and Systems*. Springer, 2010.
8. B. De Win, F. Piessens, W. Joosen, and T. Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering. In *Workshop on the Application of Engineering Principles to System Security Design*, 2002.
9. S. Franzoni, P. Mazzoleni, S. Valtolina, and E. Bertino. Towards a fine-grained access control model and mechanisms for semantic databases. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 993–1000. IEEE, 2007.
10. V. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. *NIST Special Publication*, 2014.
11. S. Jahid, C. A. Gunter, I. Hoque, and H. Okhravi. MyABDAC: compiling XACML policies for attribute-based database access control. In *Proceedings of the first ACM conference on Data and application security and privacy*, pages 97–108. ACM, 2011.
12. J. Moeys and M. Decat. Simple Tree-structured Attribute-based Policy Language (STAPL). https://github.com/stapl-dsl, 2015. [Online; accessed 2-Oct-2015].
13. S. Mutti, E. Bacis, and S. Paraboschi. SeSQLite: Security Enhanced SQLite: Mandatory Access Control for Android databases. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 411–420. ACM, 2015.
14. OASIS. eXtensible Access Control Markup Language (XACML) Standard v3.0. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf, 2013.
15. L. Opyrchal, J. Cooper, R. Poyar, B. Lenahan, and D. Zeinner. Bouncer: Policy-Based Fine Grained Access Control in Large Databases. *International Journal of Security and Its Applications*, 2011.
16. B. Parducci and H. Lockhart. XACML v3.0 Core and Hierarchical Role Based Access Control (RBAC) Profile. *OASIS*, 2010.
17. S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*. ACM, 2004.
18. A. Roichman and E. Gudes. Fine-grained access control to web databases. In *Symposium on Access control models and technologies*. ACM, 2007.
19. F. Turkmen and B. Crispo. Performance evaluation of XACML PDP implementations. In *Workshop on Secure web services*. ACM, 2008.
20. F. Turkmen, J. Hartog, S. Ranise, and Z. N. Analysis of XACML Policies with SMT. In *4th Conference on Principles of Security and Trust (POST)*, 2015.